Embedded Linux RFID Scanning Device

Team "What Is This Item" (WITI)
Santiago Gomez & Ben Kuter
santi09 / bmkuter
github.com/sgomez14/ec535final

Abstract

Our final project is an RFID scanner that can read/write messages associated with unique RFID tags. We were inspired to do this project by a commercial product that allows the blind community to scan items in their daily life. Affectionately, we call our device the "What Is This Item" - Scanner.

Our system uses an RFID sensor module, which detects RFID tags with unique IDs. In our circuit, we have a logic level converter that changes the 3.3V signal from the Beaglebone to a 5V signal for the senor, and vice versa. We employ UART serial communication between the Beaglebone and the sensor.

Our device is run entirely by a GUI userspace program developed with Qt. The GUI has elements for displaying current mode, tag ID, tag message, and typing a new message. Three buttons control the device: a read button initiates reading a tag; a write button enables saving new messages to a tag; an off button turns off the program.

1. Introduction

Ben and I set out to design a device that scans and identifies items in one's daily life. We were motivated by a blind Youtuber who showcased a product that allows her to navigate her daily routines by scanning items with special tags. We reasoned we could simulate her device by pairing the Beaglebone Black with an off-the-shelf RFID sensor and tags. To control the sensor, we developed a touchscreen intuitive GUI. A user presses READ to read a scanned tag and presses WRITE to write a new message to a tag. Given that the main interface is a GUI, our final product is not ready for the visually impaired community. But, we felt this project is a perfect platform to learn about engineering products that are universally accessible.

Matching tags and messages is a streamlined process. Our device scans RFID tags with unique IDs and matches those IDs to an internal database. When a match occurs, the device displays the message associated with the tag. Otherwise, the GUI will indicate that the tag is not in the database. A user can then enter WRITE mode to associate a new message with a scanned tag. Tag messages are written to a hashtable within the userspace program and not the memory on the tag. When a user pushes the OFF button, the program saves the hashtable to a text file. When the application starts, it reads the text file and loads the tag data back into the hashtable. Through this mechanism, our device is able to maintain continuity between sessions.

This functionality is supported by various hardware components. First is the LCD touchscreen. The small 4" x 2" screen provides a familiar interface that is expected of most mobile devices. A keyboard enables a user to enter their messages into the system. We deployed a Parallax Serial RFID Reader/Writer sensor with accompanying RFID tags. The Parallax sensor interfaces with the Beaglebone Black via a logic converter module that converts the Beaglone's 3.3V

output to 5V. The logic converter also steps down the RFID module's 5V to 3.3V so that signal can be safely ported back into the Beaglebone.

Driving the entire device is software written in C++ with the aid of several C libraries for serial communication. The GUI was developed with the Qt Framework. Our rfid_gui class is the linchpin for our software. It has methods for initializing the RFID sensor, reading the sensor, and managing various events within the GUI.

2. Design Flow

Figure 1 below outlines our system's architecture. On the software side, the userspace application processes input from a text file, the sensor, the keyboard, and tap-events on the screen. For hardware, the keyboard and touchscreen LCD connect directly to the Beaglebone. The logic level converter converts signals between the Beaglebone and the RFID sensor. The RFID tags are placed directly over the sensor to register a scan.

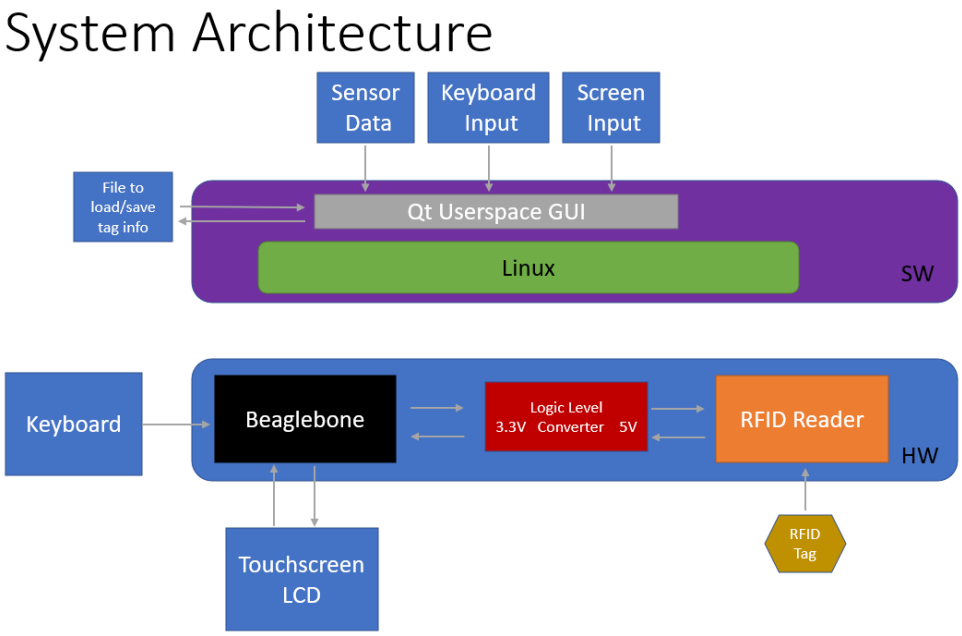

Figure 1. System Architecture of the SW & HW

Distribution of Work:

- Wiring circuit and interfacing sensor with Beaglebone - Ben
- Coding driver function for sensor - Ben
- Hardware/Software debugging - Ben & Santiago
- Data Structure Implementation - Santiago
- GUI Design and functions for processing input - Santiago

Overall Contributions:

Ben = 50% , Santiago = 50%

3. Project Details
a. RFID Module

Paramount to our design was the RFID module itself. Developed by Parallax, the 28440 RFID Read/Write Module is a TTL-level serial interface device for communicating with passive RFID transponder tags [1]. We are only concerned about reading tags for our project.

On our current setup, the Beaglebone is connected to the RFID scanner through serial terminal port 4, `/dev/ttyS4`. The exact circuitry and configuration will be shown in the next section. Here we will be discussing the process of setting up our RFID module within our userspace program. To do so, we needed to discover the correct library. We found several references via previous RFID and UART projects to the termios library[2] as well through several stackoverflow posts mentioning the library in passing.

Termios is a newer API used for terminal I/O. Specifically, it allows for opening and connecting a serial device through the standard open() call[3]. Serial devices are especially interesting because the termios attributes are written to the terminal device itself. So once a program is finished with the serial device, any attribute changes made to the terminal during that program will persist[4]. Knowing this, it is crucial that we remember to set any crucial variables for our program immediately. We can not presume the terminal starts in some default state.

First we must declare several termios variables. These include our devices baud rate, `speed_t baud_rate`, which according to the manual is 9600 baud[1]; a termios structure, `struct termios settings`, for reading and setting the device attributes; and a terminal device name as a file location, here `/dev/ttys4`. Now we open the terminal device for read/write using the aforementioned device name with the `O_RDWR` flag. We assign the resultant file descriptor to a variable, which will then be used by the termios functions.

Now we have to consider our terminal device's previous state and attributes. We use the `tcgetattr()` to initialize the termios structure we made earlier with our terminal's current attributes. Next we want to set up our termios structure in raw mode, which turns off input processing and gives us unmodified access to the terminal. This function seems to be a wrapper for setting a specific configuration of flags[4]. On top of these flags, we also want to set the `CREAD` control flag to enable the receiver. With all of our attributes defined, we now want to apply them to our terminal device. This is done through `tcsetattr()`. We must consider when
to make these changes. If the terminal device is currently in use and we change the settings immediately, the device could crash. That would just be rude of us! Instead we want to wait until any currently queued output on the terminal device has been written. However we're fine cutting in line, so we flush any queued inputs. We do this with the `TCSAFLUSH` flag for the second argument. This allows any other programs who are using the terminal device to finish operating, and then adjust the device to our own parameters. From here our terminal device is set up and ready to go. Now we need to actually access our RFID module.

By referencing the sensor's documentation we know that we need to explicitly issue commands to the sensor through `write()`. For our program, we only care about sending a read-command. We made a string containing our command, which per the documentation is a three-byte string of "`!RW`" followed by a single-byte command depending on the desired function. For `RFID_Read`, this is `0x01`. Through troubleshooting we found that unless we sent the ASCII space character (0x20), our write function would fail. So our final read command was

`{'!','R','W','SOH','SPACE'}`.

We then read the device with `read()`. The device returns 5 bytes: 1 leading status byte, and 4 data bytes. We compare the status byte to `0x01` to ensure our device has returned a RFID tag number and then parse the status and data bytes into separate strings. We recast the 4 byte data string into an integer by casting the character array to an unsigned integer pointer, then dereferencing it. The tag ID is then written to the corresponding element. If there are any errors upstream in the scanning process, nothing is written to the corresponding element and the RFID_read function is called again. To ensure transmission, the sensor manual suggests polling the device repeatedly until it returns a valid ID. The physical properties of the tags make them somewhat cumbersome and unreliable, regardless of code quality. To avoid an infinite loop we added a counter inside the scanning-loop that would cause the function to return with an error after 100 attempts. The sensor itself times-out after ~1.5 seconds and returns its own error code. This number seems high, but in practise is quite safe. It is high enough to allow for someone to not feel rushed when using the device, but small enough where any write-error with the device is detected and resolved nearly instantly.

b. Circuitry

As mentioned earlier, our Beaglebone is connected to the RFID scanner through `/dev/ttyS4` using pins P9_11 for receiving data from the sensor (along UART4_RXD) and P9_13 for transmitting data to the sensor (along UART4_TXD). Figure 2 shows the complete circuit diagram. On the RFID scanner's end, serial input (SIN) is connected to the transmission end of the Beaglebone on P9_13 and the serial output (SOUT) is connected to the receiving end of the Beaglebone on P9_11. Output from the Beaglebone goes to the input of the sensor, and the output from the sensor goes to the input of the Beaglebone. Power for the sensor was handled through our 5 volt rail, which was connected to P9_08 (SYS_5V). As a TTL-level device operating at 5 volts, compatibility with our device is immediately an issue. Our Beaglebone Black has GPIO pins operating at 3.3 volts. Directly connecting the sensor to our device may work, but we felt it best to follow the manual's instructions and use 5 volt logic. If one consults the Beaglebone Green documentation, it explicitly mentions not connecting 5 volt logic level signals to these pins for risk of damaging the board[5]. We also didn't want to risk destroying BU's property.

We acquired a Sparkfun logic level converter (LLC) from the department[1], and soldered on header pins. The high-voltage portion of the LLC was wired directly to the 5 volt bus on our breadboard. The low-voltage portion was wired to the 3 volt bus and connected to P9_04 (VDD_3v3). We then connected our data lanes to either side of the LLC, with the Beaglebone on the low-voltage side and our sensor on the high-voltage side.

While not as important in this circuit as they may be in an amplifier, consistent power delivery is always important. To facilitate constant power supply, one 100 nanoFarad ceramic capacitor was connected between the supply and ground for both the 5 volt and 3 volt rail. This will ensure a more constant voltage on either rail. In figure 2, these are shown as orange lines connecting the buses.

[1] Thank you Anthony!

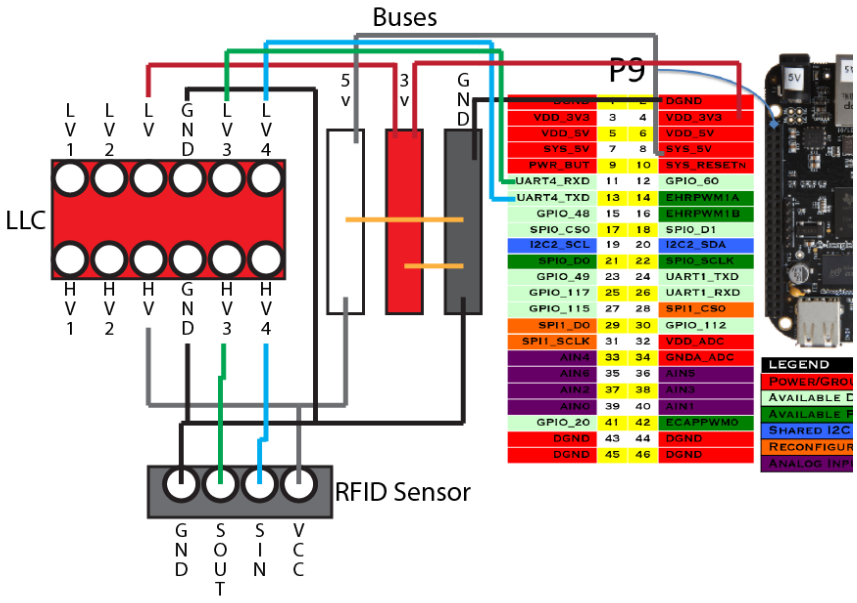

Figure 2. Our circuit diagram connecting our RFID sensor to the Beaglebone Black through a logic level converter. Note the decoupling capacitors in orange connecting either source bus to ground.

c . RFIDGui() Constructor

The software has one main class called RFIDGui. The constructor, `RFIDGui()`, makes calls to several functions that initialize various critical components of the device. First, it calls `init_serial_port()`, which establishes communication with serial port `/dev/ttyS4`. Next, a call is made to `init_tag_data()`, which reads in the text file with tag information and creates the hashtable for managing the data. Lastly, the constructor calls `init_read_mode()`, which places the device into read mode and modifies the GUI elements that are present in read mode. In Figure 4, you can see the initial state of the GUI when the program starts.

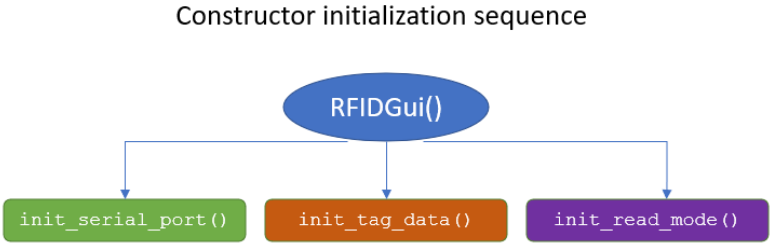

Figure 3. Constructor Init Sequence

Figure 4. Initial State at Program Start

d. READ Button

Tapping “READ” starts the reading sequence, which involves calling `read_RFID_scanner()`. The tag’s message appears in the “Scanned Message” box upon a successful scan. If the user does not place a tag, then the scanner times-out and a message appears indicating the error. Figure 5 outlines the read sequence work-flow.

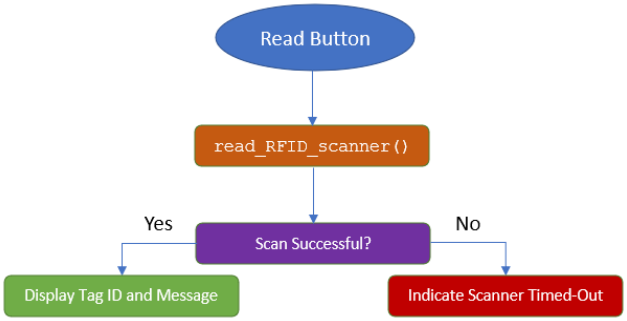

Figure 5. Read Button Work-flow

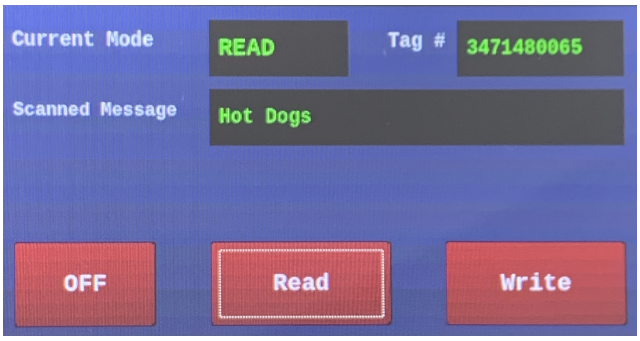

Figure 6. GUI State after Successful Read

e. Write Button

Tapping “WRITE” switches the GUI into write mode and starts the write sequence, as shown in Figure 6. The GUI displays a message instructing the user to scan a tag. Next, a call is made to `read_RFID_scanner()`. When the scanner reads the tag, a text box appears so the user can type a message with their keyboard. The textbox appearing also indicates that the tag read was successful. Figure 7 shows the GUI view at this stage. If the scanner times-out or the scanning fails for some reason, then the GUI indicates that the scan did not work. Assuming the scan succeeded, the user next types a message and presses “Enter” on their keyboard. The user’s message is saved to the hashtable and written into the text file. If the user taps the “Enter” key without a message, the GUI instructs the user to type a message.

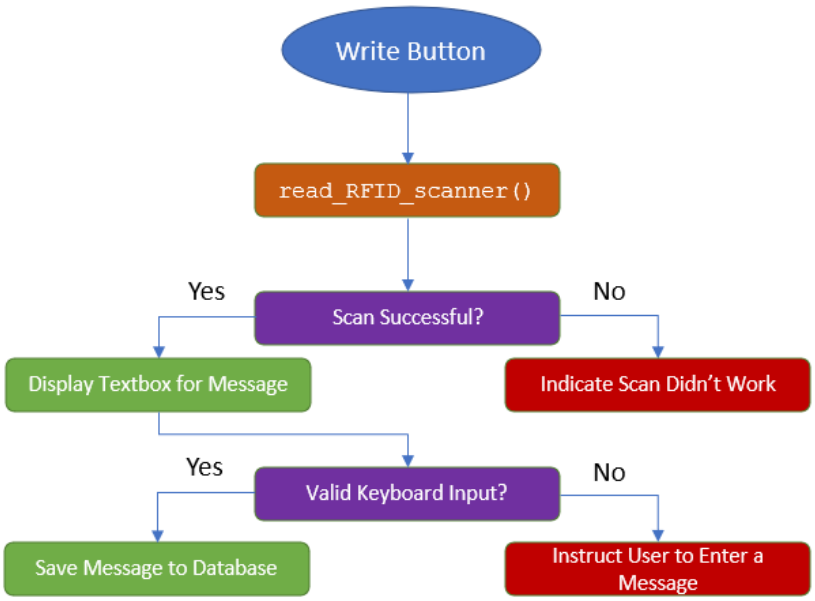

Figure 6. Write Button Work-flow

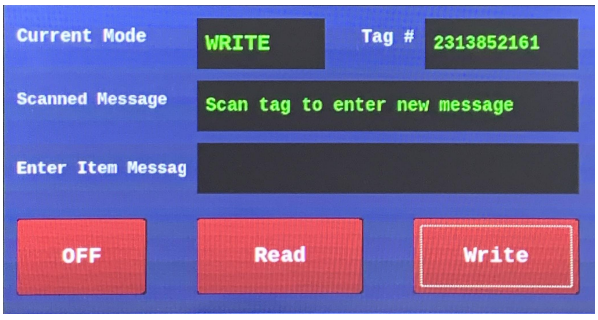

Figure 7. GUI Write State

4. Summary

To conclude, we designed and prototyped an embedded Linux RFID scanner device. Our software is written in C++ and uses the Qt Framework for developing GUIs. The GUI has very low latency and all modules function as expected.

Unfortunately we did uncover a persistent bug in our code. After the Beaglebone is turned on, the first initialization of our program does not work. The GUI launches as expected, but when the user tries to READ or WRITE a tag the program is unable to write any commands to the RFID module. We have added a failsafe to prevent this bug from bricking the Beaglebone by setting a maximum number of write-attempts to the module. The GUI displays an error message and allows the user to safely quit. On any following initializations of the program, everything works as designed.

Outside of this edge case, our product provides the necessary functionality for a user to manage their RFID tags. This device enables all users to easily tag and scan items in their daily lives. The next iteration of our project will have input and output with audio and haptics. These two new features will provide significant value to the visually-impared community since the device will give them more autonomy over their daily lives.

References

[1] Parallax. “RFID Read/Write Module, Serial (#28440)” https://www.parallax.com/package/rfid-read-write-module-product-guide/
[2] Bryant Moquist, Ian Bablewski, Calvin Flegal. RFIDTransactionUnit. https://github.com/bmoquist/RFIDTransactionUnit
[3] Serial Programming/termios. https://en.wikibooks.org/wiki/Serial_Programming/termios
[4] GNU C Library Reference Manual. https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_17.html
[5] Seeed Studio. Beaglebone Green Documentation. https://seeeddoc.github.io/Beaglebone_green_wireless/
[6] GUI CSS Presets: Adaptic. https://qss-stock.devsecstudio.com/